UCRL-WEB-201481

Gang Scheduler User Guide

Table of Contents

Preface

Scope: LLNL's gang scheduler supports the concurrent scheduling of processors for parallel programs. This guide tells how to prepare both C and Fortran programs so they will invoke the gang scheduler appropriately (initialization, program registration, and process registration), how to monitor gang-scheduled jobs by running XGANG, how to interpret XGANG output, and how to handle special (error or query) situations. The basic job-management approach used by LLNL's gang scheduler is also explained, along with the role of job preemption in gang scheduling.

This manual is adapted and expanded from parts of UCRL-MI-129060 and UCRL-JC-129927, both by Moe Jette.

Availability: Versions of LLNL's gang scheduler can be invoked (indirectly) by programs running on machine GPS320 (a 32-node member of the Compaq GPS cluster), as well as on LC's open and secure IBM SP machines (such as Blue and White). These versions are developmental and may vary between operating systems. DPCS support for gang scheduling across all machines in a Compaq cluster ended with DPCS version 6.7 (spring, 2002).

Consultant: For help contact the LC customer service and support hotline at 925-422-4531 (open e-mail: lc-hotline@llnl.gov, secure e-mail: lc-hotline@pop.llnl.gov).

Printing: The print file for this document can be found at:

```
on the OCF: http://www.llnl.gov/LCdocs/gang/gang.pdf
on the SCF: https://lc.llnl.gov/LCdocs/gang/gang_scf.pdf
```

Introduction

The gang scheduler (originally for Cray computers, then generalized for other operating systems) was developed by Lawrence Livermore National Laboratory (LLNL) to provide users with a way to better harness the power of large, parallel machines.

In this document, the following special terms appear:

gang scheduling	groups a program's parallel threads of execution (next item) into a "gang," then concurrently schedules one independent processor for each thead in the gang. Such concurrent scheduling can occur on a single computer as well as across multiple computers in a cluster, and it allows a parallel program to take advanage of its inherent parallelism even under heavy loads.
thread	is a path of program execution that can proceed at the same time as others. Included as threads are processes generated by fork system calls, MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) tasks, and Pthreads.
space sharing	allows multiple programs (or threads) to execute independently on distinct processors at the same time (but not take turns on one processor).
time sharing	allows different programs to take turns running on the same processor. Gang scheduling arranges just such concurrent program access to processors and then concurrent access removal, so that each program sees dedicated computing resources during its periods of execution (memory and I/O bandwidth excluded).

Most parallel UNIX systems fail to provide synchronized compute resources for parallel programs, and even slight levels of competition for processors can severly impact program performance. Many tightly synchronized programs continuously poll semaphores at synchronization points (called "spin-wait"), rather than relinquish a processor. Unless a program's threads of execution can be concurrently "gang scheduled" for processors, this spin-wait pattern can waste substantial time without advancing a program's progress.

Usage Overview

How to Use

The relevant instructions for using the gang scheduler depend on the design of your parallel program. There are three cases:

MPICH Users If your program loads with the default MPICH library, then you can invoke gang scheduling with no code or load changes just by setting the GANG_SCHEDULE environment variable (e.g., under the C shell, by using setenv) before you execute the program. Set GANG_SCHEDULE to:

1	if your program does NOT use either of the signals SIGUSR1 or SIGUSR2 for its own purposes, or
2	if your program does use either of the signals SIGUSR1 or SIGUSR2 for its own purposes.

Digital MPI (DMPI) Users

If your program loads with Digital's MPI library (DMPI) on a Compaq machine (such as GPS320), then you can invoke gang scheduling with THREE STEPS that include code and load changes as follows:

call gs_register()
Insert this one extra function call into your code immediately after you call MPI_Init, to automatically manage your interface to the gang scheduler. The mpichk2.f Fortran example (page 29) discussed below illustrates this step.

load with /usr/local/lib/libgs.a
in addition to your usual MPI library, to make the gang scheduler support functions available to your program.

setenv GANG_SCHEDULE 2
before you execute the program (to invoke gang scheduling in a way that frees signals SIGUSR1 and SIGUSR2 for other uses with Digital MPI).

PVM Users and others wanting customized control

If your program uses the PVM approach or if you want to manipulate the gang-scheduler invocation routines yourself, then you can add gang scheduling support directly to your program by installing the "application program interface" (API) features described in the rest of this document and illustrated in most of the examples (page 29) below. The steps (details in other sections) are summarized here:

- Your application program must build a data structure describing its resource requirements (eg. 3 CPUs on computer east and 4 CPUs on computer west), call a function (GangJobRegister (page 10) for C, GangResourceRegister (page 27) for Fortran) to register these resource requirements, and get a job ID.
- Your program must then register (using GangProcAdd (page 14)) its processes, process group(s), or sessions as part of the job using the whole-job ID returned at resource-registration time.
- To provide it with needed gang-scheduler data-type and function definitions, your program must include (be compiled with) either the C header file GangUserAPI.h (or the thread-safe equivalent GangUserAPI_r.h) or else the Fortran header file fGangUserAPI.h (or the thread-safe equivalent fGangUserAPI_r.h). All these header files reside in the directory /usr/local/include.
- You must load your program with its usual libraries plus /usr/local/lib/libgs.a (or its thread-safe equivalent library libgs_r.a), which includes the gang scheduler API functions.

In all three cases above, the gang scheduler notes when processes terminate and performs the necessary clean-up automatically.

When to Use

The benefits from gang scheduling are achieved by synchronized scheduling of CPU and memory resources required by the parallel program, and by dedicating CPUs to the job's processes. The overall rate at which resource are made available to the program is roughly the same either with or without gang scheduling, however. Change in program performance depends upon several factors, including: level of synchronization required, cache reference patterns, and memory reference patterns. Additionally, the current load on the machine has a major impact on the performance of gang-scheduled jobs. Some programs demonstrate speedups as much as 700%. Other programs have been seen to actually run slower due to memory contention. Typical speedups for complex applications have been in the 5% to 50% range.

The gang scheduler is not appropriate for use on all parallel jobs. Communication with the gang scheduler daemon(s) takes a few seconds for initialization, making it best suited for longer running jobs. Jobs without tightly synchronized communications or significant I/O may operate well without gang scheduling.

Fortran Users Note:
The function descriptions in the main sections of this document are oriented toward C programmers. A separate later section (page 27) has been prepared for Fortran programmers with descriptions of easy to use subroutines for that environment. Those Fortran calls are patterned after the C functions described below. It is advisable that you read the analysis in the main text without concerning yourself with the syntax of the C subroutines calls, then look in the Fortran section (page 27) for details on the actual calls you will need to use.

Signals

The signals SIGUSR1 and SIGUSR2 are (normally) used to improve the overlap achieved by the gang scheduler.

The gang scheduler creates a "class" for each registered parallel program on every computer the program will use. When processes are registered as a component of the program (details below), their process IDs are added to the class. To stop the program (at the end of a time slice) its class is allocated zero resources, but this may not be completely effective if idle processors exist on the computer. To more effectively stop the program, the SIGUSR1 and SIGUSR2 signals are optionally used to pause and continue its execution. Your program can explicitly disable gang-scheduler use of these signals if they are required for other purposes, but doing so will reduce concurrency and may reduce program performance too. These signals also permit a more tightly synchronized stopping of the program at the end of a time slice than can be achieved by class scheduling alone.

If your program makes use of these signals for other purposes, the gang scheduler will alter the program's behavior unless you explicitly disable them (see the GangJobRegister (page 10) or GangJobClass (page 24) sections below for details; a Fortran version of this call is also available). MPI users may disable signals by setting the GANG_SCHEDULE environment variable to 2. To avoid establishing extraneous signal handlers, disable signals either in the GangJobRegister function call or prior to this by specifying a NULL job ID in a call to GangJobClass. (page 24)

Disabling the SIGUSR1 and SIGUSR2 signals may reduce your program's overlap by an amount which varies with the system workload. Future modifications in the operating system (for example, slated for Digital Unix 5.0) will eliminate gang-scheduler use of these signals.

Process Initialization (Obsolete)

The former use of routine GangJobInit by each process to initialize for gang scheduling on Cray computers running UNICOS is *not* required for any other platform or operating system. GangJobRegister (next section) is now the first gang scheduling routine to invoke.

Parallel Program Registration

The GangJobRegister function (in C) performs two operations:
(1) It registers the program's resource requirements with the gang scheduler (in Fortran, you must use a separate function called GangResourceRegister (page 27), described later in the Fortran section), and
(2) it returns a job ID. This job ID is used in all subsequent communications with the gang scheduler.

This is by far the most complex function to communicate with the gang scheduler as detailed below. Until a job and its processes are registered with the gang scheduler, each process (and its tasks) are scheduled independently by the operating system. Therefore parallel jobs should register at part of their initialization to optimize performance.

The GangJobRegister syntax (in C) is:

```
extern gsRetVal GangJobRegister(
        struct GangJobId *gang_job_id_ptr,             /* The job ID    */
        JOB_CLASS job_class,                           /* The job class */
        struct GangResources *gang_resource_list[]);   /* The resource list */
```

The return code is gsSuccess if no errors were encountered. Otherwise, an error code is returned.

GangJobRegister Arguments

gang_job_id_ptr

is a pointer to the job ID structure. On the first call, this structure should be null filled. A non-null job ID structure implies that the call is intended to update resource requirements for a previously registered job. If a zero CPU requirement is specified, the job ceases to be controlled by the gang scheduler.

job_class specifies the job's class. Valid job classes at LLNL are CLASS_EXPRESS, CLASS_INTERACTIVE, CLASS_BATCH, CLASS_BENCHMARK, CLASS_STANDBY, and CLASS_UNIX. Different job classes may be used at other sites, but modifications to the source code are required to add additional job classes. New job classes could be restricted to certain users, have limits on their CPU count, etc.

CLASS_EXPRESS is available only to the system super user for jobs requiring the best throughput possible.

CLASS_INTERACTIVE

is for interactive jobs and is designed for short running program development work. Jobs of this class may be preferentially scheduled. Jobs submitted as CLASS_INTERACTIVE will be reclassified as CLASS_BATCH after executing for more than 60 CPU minutes (configurable by system administrator, see MAX_TIME_INTERACTIVE). This permits the system to provide optimal interactivity to shorter jobs requiring rapid throughput.

CLASS_BATCH is for batch jobs. It is suitable for long running production jobs.

CLASS_BENCHMARK

designates jobs that will not be preempted once started, but may take a long time to begin execution. This may be used for timing purposes.

CLASS_STANDBY specifies jobs that will be allocated otherwise unused system resources. This is suitable for low-priority work.

CLASS_UNIX specifies jobs that are registered to the gang scheduler, but not gang scheduled. This permits use of the gang scheduler user interface tools (and DPCS tools) for job tracking purposes without gang scheduling in those cases where performance is better with UNIX scheduling.

Normally, you should specify CLASS_BATCH or CLASS_INTERACTIVE. The flag NO_SIGNALS_FLAG may be ORed with other job classes to disable signals and set the job class simultaneously (as shown in the example in the next section). For another way to disable signals (by using GangJobClass), see a later section (page 24).

gang_resources_list

is a pointer to a list of pointers (example follows). Each pointer refers to the resource requirements of the program on a specific computer. The last pointer should be NULL. The resource requirements are specified in the GangResources structure, which includes the following fields:

machine	The name of the computer these requirements apply to (e.g., west).
cpu_count	The CPU count requested. The number actually allocated may be less due to system constraints. The actual number of CPUs assigned to the job will be returned in this field.
cpu_min	The minimum acceptable CPU count.
mega_mem	The memory requirements of the program in megabytes. This is optional, but may provide better performance if provided.
giga_disk	The disk storage requirements of the program in gigabytes. This is optional, but may provide better performance if provided.

GangJobRegister Example

The C code fragment below shows the use of the GangJobRegister function. This is Compaq-appropriate code. noted in comments near the beginning.

While it is possible to register resource requirements with independent calls for each computer, this will usually result in significantly lower throughput than if all resource requirements for all computers to be associated with a job are included in a single function call as shown below. Requesting more CPU resources than you are actually able to use will typically result in lower throughput than an accurate value. For example, a job requesting four CPUs will normally be allocated those resources about half as frequently as a job requesting two CPUs.

```
gsRetVal rc;
struct GangJobId my_job_id;
struct GangResources gang_resources[2];
struct GangResources *gang_resource_list[3];

/* Clear job_id on first call, otherwise, */
/* the call will apply to an existing job */
bzero(&my_job_id, sizeof(my_job_id));

/* Define resource requirements for computer gps320 */
strcpy(gang_resources[0].machine, "gps320");
gang_resources[0].cpu_count = 8;
gang_resources[0].cpu_min = 120;
gang_resources[0].mega_mem = 5;
gang_resources[0].giga_disk = 2;
gang_resource_list[0] = &gang_resources[0];

gang_resource_list[2] = NULL;

rc = GangJobRegister(&my_job_id, CLASS_INTERACTIVE | NO_SIGNALS_FLAG,
                     gang_resource_list);
if (rc != gsSuccess) {
        printf("Error from GangJobRegister %d\n", rc);
        exit(1);
}
```

Process Registration (Addition)

After the job is registered, you must use GangProcAdd to specify ONE of these:

- the individual processes associated with the job. Full support is provided when the individual processes are identified and this is the recommended mode of operation.
- the process groups associated with the job. Support is not currently provided to suspend, resume, or kill a job in which only process groups are identified.
- the sessions associated with the job. Support is not currently provided to suspend, resume, or kill a job in which only sessions are identified.

You can not mix these three ways of process registration (addition). You can not, for example, specify a process group plus individual processes.

GangProcAdd Arguments

The call to register (add) processes is GangProcAdd, with this C syntax:

```
extern gsRetVal GangProcAdd(
        struct GangJobId *gang_job_id_ptr,      /* The job ID */
        JOB_PROC proc_type,                     /* Type of process */
        int id);                                /* ID of process */
```

where the three agruments are:

gang_job_id_ptr

is the job ID which was returned by the GangJobRegister function (described in the previous section). The job's ID applies throughout the computing environment and it is your responsibility to propagate it as needed.

proc_type — indicates how components of the job will be specified. The allowed values are:

PROC_ID	specifies process IDs, the preferred method.
PROC_GROUP_ID	specifies process group IDs.
PROC_SESSION_ID	specifies session IDs, not available on CRAYs.

id — is the ID of the job's component. It is usually easiest to specify the process IDs as processes are started.

GangProcAdd Example

The C code fragment below demonstrates the use of GangProcAdd (to register individual processes using their own process IDs, the preferred method when available).

```
rc = GangProcAdd(&my_job_id, PROC_ID, getpid());
if (rc != gsSuccess) {
        printf("Error in GangProcAdd %d\n", rc);
        exit(1);
}
```

Job Monitoring with XGANG

An X-windows tool called XGANG is available for monitoring the gang scheduler. It is simple to use and each window has a Help button to describe what is displayed and the window's options. The displays are updated automatically when the gang scheduler has selected new jobs to execute (about every 30 seconds). This means that job registration, process addition, class changes, and other activities will not be visible until the next cycle during which the gang scheduler writes to its database and updates its display. The periodic update is meant to minimize system overhead in running the gang scheduler.

Running XGANG

To monitor the scheduling of a job, follow these steps:

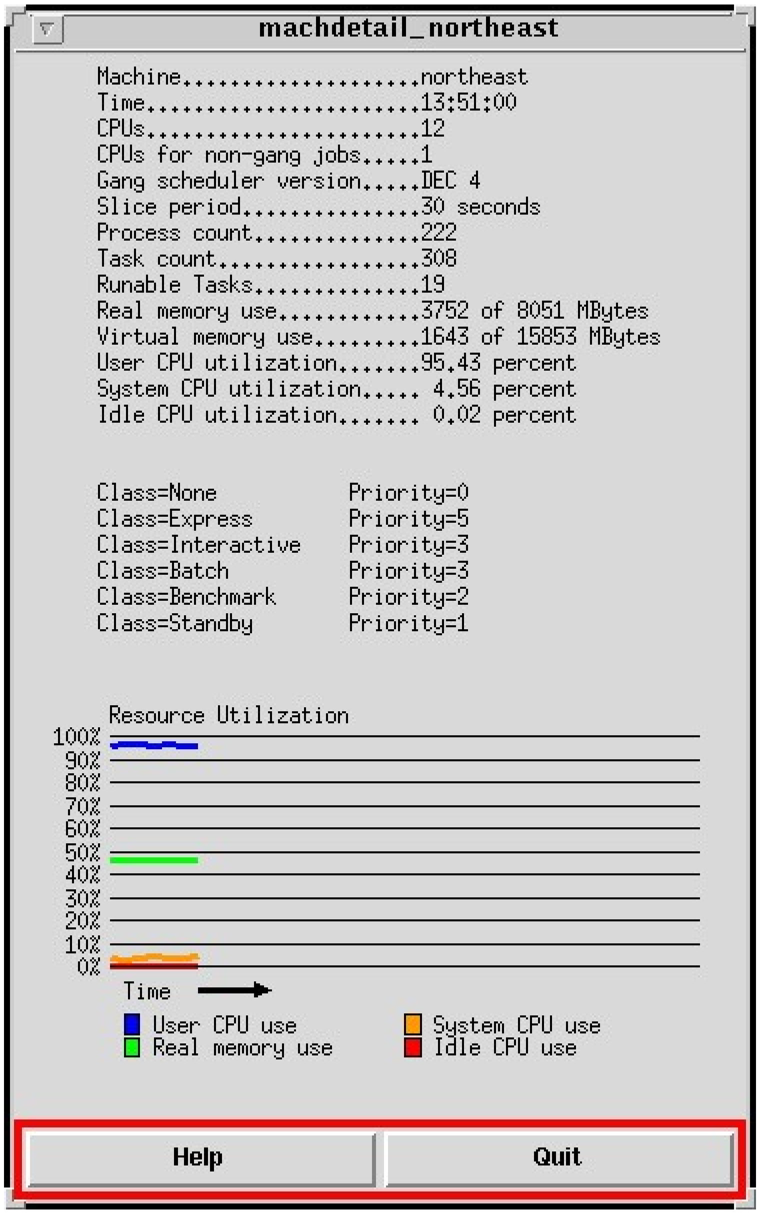
- Initiate xgang (the UNIX file name is all lowercase).
- Click on the button of one of the computers executing the job that you are trying to monitor. This will create a window with the status of that computer and a list of gang-scheduled jobs on it.
- Click on any job(s) of interest to see job details reported (example output appears below).
- If the job is executing on multiple computers, click on the Show Job On All Hosts button to see the job's details on all computers.

The colors used in the XGANG displays are configurable by means of environment variables. The environment variables that you can set are GANG_COLOR_1, GANG_COLOR_2, GANG_COLOR_3, and GANG_COLOR_4. These variables are utilized by the Tcl/Tk program and should specify a color name (blue, red, green, orange, black, white, yellow, pink, etc). The colors specified will be used for both the computer and job detail displays (illustrated in the next two sections).

The font used in the displays can be reconfigured through the use of an environment variable too. The environment variable GANG_FONT_1 will set the font used in most windows. The default font is FIXED. Alternate font values are system dependent. You may view a list of possible fonts by executing the UNIX command XLSFONTS on the machine where you plan to run XGANG. Note however that if you pick a nonfixed-width font, you will probably spoil the alignment of the XGANG output.

Display by Computer

A sample display of detailed computer information from XGANG is shown in Figure 1. Below the figure is an itemized explanation of each field that it displays, listed in the order in which they appear. (The specific machine mentioned on the example display, NORTHEAST, is no longer available at LC, but the use of XGANG remains the same.)



```
machdetail_northeast

Machine.....................northeast
Time........................13:51:00
CPUs........................12
CPUs for non-gang jobs.....1
Gang scheduler version.....DEC 4
Slice period...............30 seconds
Process count..............222
Task count.................308
Runable Tasks..............19
Real memory use............3752 of 8051 MBytes
Virtual memory use.........1643 of 15853 MBytes
User CPU utilization.......95.43 percent
System CPU utilization..... 4.56 percent
Idle CPU utilization....... 0.02 percent


Class=None          Priority=0
Class=Express       Priority=5
Class=Interactive   Priority=3
Class=Batch         Priority=3
Class=Benchmark     Priority=2
Class=Standby       Priority=1
```

Figure 1. A sample of detailed computer information from XGANG.

Machine	Name of the computer reported on.
CPUs	The count of CPUs on this computer.

CPUs for non-gang jobs	The count of CPUs reserved for non-gang scheduled jobs. The system administrator can set this to 1 or more in order to insure timely response for interactive jobs. You can only be allocated CPUs which are on the computer, but not reserved for non-gang scheduled jobs.
Slice period	The time period of each gang scheduler processor allocation.
Process count	Count of processes on this computer.
Task count	Count of tasks (or threads) on this computer.
Runable tasks	Count of tasks currently in run state (ie. competing for the available processors). This is used to computer a "fair" allocation of resources for gang scheduled jobs.
Real memory use	Displays real memory use and total real memory on this computer.
Virtual memory use	Displays virtual memory use and total virtual memory on this computer.
User CPU utilization	Displays CPU resources being consumed by user jobs of any nice value.
System CPU utilization	Displays CPU resources being consumed by the operating system and system processes.
Idle CPU utilization	Displays CPU resource not being consumed by the user or system.
Class/Priority	Displays all possible job classes and their relative priority.

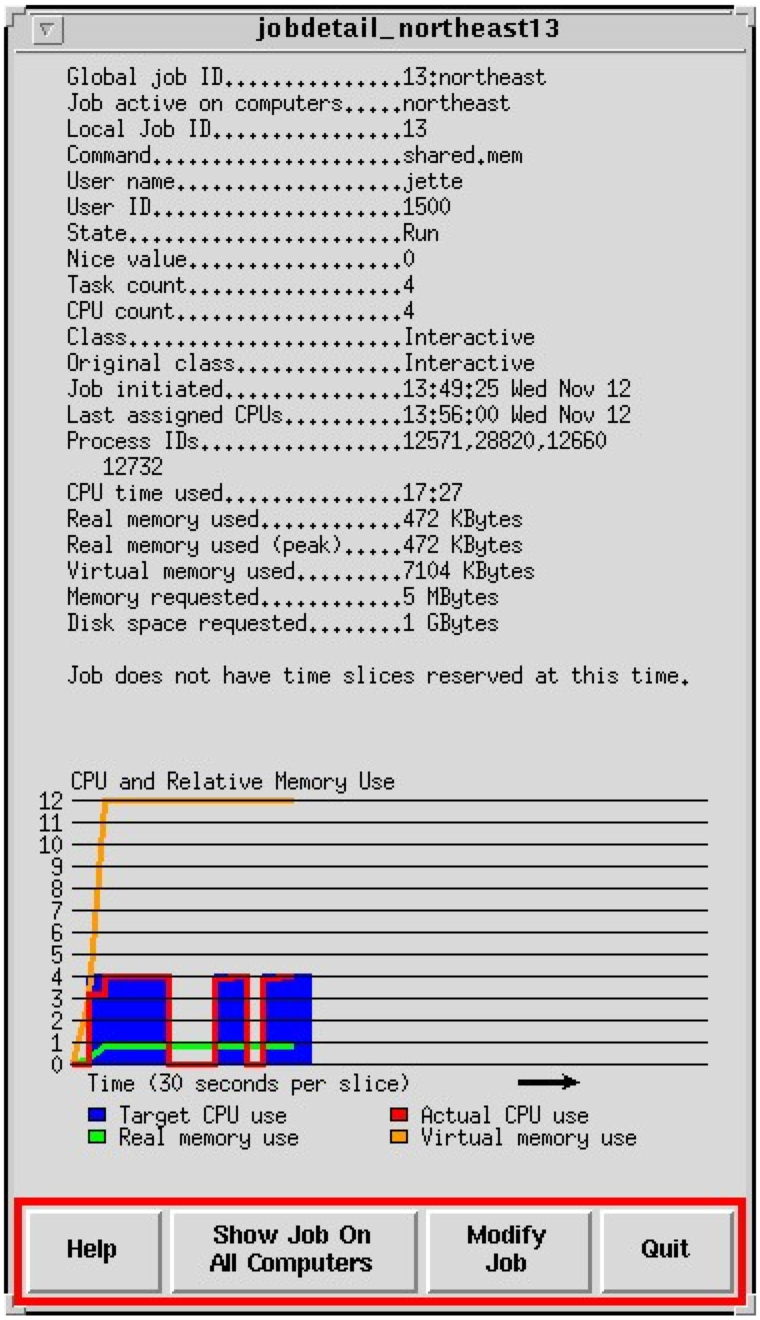
Display by Job

A sample display of detailed job information from XGANG is shown in Figure 2. Below the figure is an itemized explanation of each field that it displays, listed in the order in which they appear. (The specific machine mentioned on the example display, NORTHEAST, is no longer available at LC, but the use of XGANG remains the same.)

The "Show Job On All Computers" button at the bottom of the window enables you to see all components of the job on all computers easily. The "Modify Job" button at the bottom provides a mechanism to manage the job as a single entity, even if it spans multiple machines. Job management functions include: kill job, suspend job, resume job, and change job.

Note that some job class changes are restricted to system administrators. Also note that job class changes can be initiated by the system in response to changes in the job's nice value or the job being stopped. If the job is stopped, it will be placed into standby class and can be removed only be a system administrator. Jobs that are suspended are not subject to class changes, but may have their record purged from the gang scheduler if the job is suspended for an extended period.

Job modifications are based upon the gang scheduler's database, which is update at each time slice and might not reflect the latest situation such as newly added processes. After modifying a job's state (particularly killing, suspending or resuming a job), insure that all components of the job are addressed.



```
Global job ID...............13:northeast
Job active on computers.....northeast
Local Job ID................13
Command.....................shared.mem
User name...................jette
User ID.....................1500
State.......................Run
Nice value..................0
Task count..................4
CPU count...................4
Class.......................Interactive
Original class..............Interactive
Job initiated...............13:49:25 Wed Nov 12
Last assigned CPUs..........13:56:00 Wed Nov 12
Process IDs.................12571,28820,12660
   12732
CPU time used...............17:27
Real memory used............472 KBytes
Real memory used (peak).....472 KBytes
Virtual memory used.........7104 KBytes
Memory requested............5 MBytes
Disk space requested........1 GBytes

Job does not have time slices reserved at this time.
```

Figure 2. A sample of detailed job information from XGANG.

Global Job ID	Job ID throughout the cluster. It is composed of the name of the computer on which the job originated and a sequence number.
Job active on computers	List of computers on which this job exists.
Local Job ID	Job ID on this computer. Gang scheduled jobs existing on multiple computers may have different local job IDs on each computer.

Command	The name of one of the commands associated with the job on this computer.
User name	Name of the user who initiated the job. This is used to determine who can modify the job .
User ID	The UID of the user who initiated the job.
State	Current status of the job (Run, Wait, or Hung). A hung job is not using resources the gang scheduler is attempting to allocate to it, so the gang scheduler has stopped trying. If the job starts using resources again, gang scheduling will resume.
Nice	The lowest nice value of any process associated with this job. This is used for scheduling purposes.
Task count	Number of tasks (or threads) associated with this job.
CPU count	Number of CPUs to be associated with the job whenever in Run state.
Class	Current class of the job (Express, Interactive, Batch, Benchmark or Standby).
Original Class	Original class of the job. The current class of a job may change due to system administrator intervention or nice value changes.
Job initiated	When the job was registered with the gang scheduler.
Last assigned CPUs	When the job was last assigned CPUs (in Run state). A job will be purged if it has not been assigned CPUs for "too" long, that period being configurable by the system administrator.
Process IDs	The process IDs associated with this job. (Only one of process, group, or sessions IDs will show.)
Process group IDs	The process group IDs associated with this job. (Only one of process, group, or sessions IDs will show.)
Session IDs	The session IDs associated with this job. (Only one of process, group, or sessions IDs will show.)
CPU time used	The cumulative CPU time consumed by those processes currently associated with this job on this machine. If a process terminates, its cumulative CPU time ceases to be reported here. If the actual CPU use is significantly below the CPU allocation, the job may require tuning in order to achieve the desired level of parallelism. Your job's throughput is normally best if the actual CPU use equals the requested CPU count. If there is a significant difference you should consider lowering your CPU count

requested in order to be allocated a smaller number of CPUs more frequently for a net increase in throughput.

Real memory used

Real memory being used by all processes currently associated with this job. If you see wide variations in real memory use between job run and wait states, the job is paging out to virtual memory and you should notify your system administrator to inspect the gang scheduler parameters for possible tuning problems.

Virtual memory used

Virtual memory being used by all processes currently associated with this job.

Memory requested

Megabytes of memory space being requested by this job. This information is optional, but may effect job scheduling.

Disk space requested

Gigabytes of disk space being requested by this job. This information is optional, but may effect job scheduling.

Advanced Topics

Computer Failure

If one of the computers being gang scheduled fails, a portion of your program may cease execution. You should probably add logic to your own program to deal with the failure of one computer in a cluster in the way that you prefer. The gang scheduler will attempt to continue scheduling of any remaining components of your program.

Error Code Translation

Error codes can be translated into a descriptive message using the GangErrMsg function. Its only argument is the gang scheduler error code and it returns a string as demonstrated below:

```
printf("Gang Scheduler error code %d: %s\n",rc, GangErrMsg(rc));
```

Process Removal

Processes added to a gang scheduler job can be explicitly removed from that job. This is normally accomplished when the process no longer exists, but can be done explicitly by the GangProcRemove call. This call's arguments are identical to those of GangProcAdd and are detailed below:

```
extern gsRetVal GangProcRemove(
        struct GangJobId *gang_job_id_ptr,   /* The job ID */
        JOB_PROC proc_type,                  /* Type of process identifier */
        int id);                             /* ID of process etc. */
```

A job and all of its processes can explicitly removed from the gang scheduler's control by registering for zero CPUs on each computer being used.

Job Class Change

The class of a job can be modified using the GangJobClass function. Note that some job classes have restricted availability and there are some restrictions on movements between job classes.

```
extern gsRetVal GangJobClass(
        struct GangJobId *gang_job_id_ptr,         /* The job ID */
        JOB_CLASS job_class);                      /* New job class */
```

GangJobClass can also be used to disable the use of SIGUSR1 and SIGUSR2 by the gang scheduler by issuing the call with a job_class of NO_SIGNALS. To insure that no signal handlers are established by the gang scheduler and no signals are sent to a job, execute "GangJobClass(NULL,NO_SIGNALS);" prior to executing GangJobRegister. Alternately you may include the flag NO_SIGNALS_FLAG in the job class's value as shown:

```
GangJobRegister(&my_job_id, CLASS_BATCH | NO_SIGNALS_FLAG, &resource_list).
```

If GangJobClass is executed later in a job, the signal handlers will continue to exist, but signals will cease to be sent to the job by the gang scheduling system. See the GangUserAPI.h file for information about other job classes. Other than to disable signals, the use of this call by users is not advised.

GangHostQuery

You may wish to assess the availability of resources on a machine prior to attempting to change resources. The GangHostQuery can be used to determine what proportion of gang scheduler time slices would be available to this job if the specified additional CPU resources were applied to the problem. Note that the number of additional CPUs is specified rather than the aggregate number of CPUs to be applied to the problem. Normally applying additional resources to the problem will result in those resources being provided at less frequent intervals.

```
extern int GangHostQuery(
        char *hostname,      /* Computer where resources are requested */
        int cpu_count);      /* Number of additional CPUs to allocate */
```

Specify the name of the computer on which these CPU resources are desired and the CPU count. The function will return the percentage of current time slices which would be applied to the problem. A value of zero indicates that no additional resources can be applied at present. A value of 100 or more indicates that additional CPU resources can be applied without reducing the frequency at which those resources are applied. Note that this call will not reserve resources for the job and that changes in the workload occur frequently. When a request is actually issued to reserve those resources, the quantity of resources available may be more or less than earlier reported by this function.

Remote Procedure Calls

Event Logging

A remote procedure call (RPC) is provided for gang scheduler developers to log events directly into the gang scheduler's log. The use of this call by users is not advised, but is detailed below for completeness:

```
extern gsRetVal GangMessage(
        char *message,          /* The message to log */
        char *host);            /* Computer on which to log the message */
```

Timing Information

Timing information is available through a second RPC. Since the information returned is based upon the gang scheduler daemon's database, the precision of the data is limited to that of the time-slice duration. The times reported are the CPU time allocated and CPU time used. The CPU time allocated is the product of the program's CPU count and the time-slice duration summed over all time-slices and all computers used. The CPU time used is the CPU time consumed by all processes currently associated with the program.

WARNING:
The CPU time used by processes that have already terminated prior to the execution of this RPC is not reported. Both times are in units of seconds as detailed below:

```
extern gsRetVal GangGetStats(
        struct GangJobId *gang_job_id_ptr,  /* The job ID */
        long int *seconds_allocated,        /* CPU sec allocated to job */
        long int *seconds_used)             /* CPU time used by job */
```

Thread Use

Since all threads are associated with a single process, multithreaded programs only need to register a single process. There is no need to register each thread for gang scheduling. Because of a temporary dependence upon signals in the current implementation and complications in signal handling within a threaded program, overlap will vary somewhat with the system's workload.

For threaded programs, always use the separate, thread-safe header files (GangUserAPI_r.h for C, and the equivalent fGangUserAPI_r.h for Fortran) as well as the corresponding thread-safe support library (libgs_r.a).

To improve thread support, some former functions (in the thread-safe library ONLY) have been replaced by Fortran subroutines. Each subroutine now has an (additional) integer argument in which to place the return value (instead of simply having the function return it). Three calls are affected by this library update:

OLD: *return_value* = GangJobClass(*f_job_id*,*f_job_class*)
NEW: call GangJobClass(*f_job_id*, *f_job_class*, *return_value*)

OLD: *return_value* = GangJobRegister(*f_job_id*)
NEW: call GangJobRegister(*f_job_id*, *return_value*)

OLD: *return_value* = GangProcAdd(*f_job_id*)
NEW: call GangProcAdd(*f_job_id*, *return_value*)

Fortran Use

A subset of the subroutine calls described above has been prepared for ease of use from the Fortran programming environment. The general usage pattern is the same as for C, except that in Fortran two separate functions register the job resource needs (GangResourceRegister) and get its job ID (GangJobRegister), while in C the second function performs both of these tasks together. This section ends with a simple Fortran code example.

First the storage for the gang scheduler job ID must be created. The header file fGangUserAPI.h contains the size of the structure in the integer GANG_JOB_ID_LEN. The job_id should then be created as an array of GANG_JOB_ID_LEN elements of type integer*4.

GangJobIdClear:
The GangJobIdClear subroutine will clear the initial job ID variable, which is the only argument to the subroutine. The job ID must be cleared prior to registration with the gang scheduler or an attempt will be made to apply the request to an existing job.

GangResourceRegister:
Next register for the job's resource requirements on each computer to be used with the (Fortran-only) GangResourceRegister subroutine. The GangResourceRegister subroutine in Fortran has the same five arguments as does the gang_resources_list of pointers in the C version of GangJobRegister (detailed in the GangJobRegister Arguments (page 10) section above), namely: name of the computer, number of CPUs desired on that computer, minimum number of CPUs acceptable on that computer, megabytes of memory storage desired (optional), and gigabytes of disk storage desired (optional). See the code example below.

GangJobRegister:
After all of the resource requirements have been specified for all computers to be utilized, issue the GangJobRegister call to register the job and get a job ID back. This job ID will be used in calls to the GangProcAdd subroutine, which associates the process ID of the calling process with job. The job's ID applies throughout the computing environment and it is your responsibility to propagate it as needed. All of the processes associated with this job must register with the same job ID.

GangJobClass:
A job's class or signal handling may be altered with the call GangJobClass, which takes the job ID and new class as arguments. The job class NO_SIGNALS will disable use of SIGUSR1 and SIGUSR2 by the gang scheduler. Other job classes are as described in the C-language GangJobRegister section above. (page 10) For the job class to be in effect over the entire lifetime of the job, execute GangJobClass between GangJobIdClear and GangJobRegister. See the GangUserAPI.h file for information about other job classes.

GangJobTime:
The GangJobTime subroutine will return the CPU time allocated and used, if you want timing information.

The code fragment below shows the code required to register a job to run with 8 CPUs on GPS320. The process is also registered as part of that job. For all of these calls, any errors will result in an error description being printed and the program terminating. The GangJobClass, GangJobRegister, and GangProcAdd functions return error codes.

```
      include 'fGangUserAPI.h'
      integer*4 gang_job_id(GANG_JOB_ID_LEN)
      real seconds_allocated, seconds_used

c     Register the job's resource requirements and get a job id
      call GangJobIdClear(gang_job_id)

c     Disable signals, note value of gang_job_id is not set yet
c     DISABLEING SIGNALS IS NOT RECOMMENDED AS A DEFAULT
      call GangJobClass(gang_job_id, NO_SIGNALS)

      call GangResourceRegister("gps320", 8, 1, 3, 6)
      i = GangJobRegister(gang_job_id)
      if (i .eq. 0) go to 150
      write(6,*) 'GangJobRegister error ',i
      stop
 150  continue

c     Register the processes which are part of the job
      i = GangProcAdd(gang_job_id)
      if (i .eq. 0) go to 160
      write(6,*) 'GangProcAdd error ',i
      stop
 160  continue

c     Do some work
      call work(gang_job_id)

c     Get the timing information
      call GangJobTime(gang_job_id, seconds_allocated, seconds_used)
      write(6,*) 'GangJobTime: seconds_allocated ',seconds_allocated
      write(6,*) 'GangJobTime: seconds_used      ',seconds_used
```

Examples

These sample parallel programs (OCF only) were developed by LC's Distributed Computing Tools Group (DCTG) to show how to install the gang scheduler API features into a variety of programming situations. Each example description here links to the corresponding source and MAKE files, which reside in other DCTG directories. Remember, however, that (as the How to Use (page 5) section above reveals), only PVM users must take this complex approach; MPICH and Digital MPI users can invoke gang scheduling WITHOUT making these elaborate code changes (and sample mpichk2.f shows the simplier alternative that most DMPI users will prefer).

Fortran MPI (one host)

Language: Fortran
Sample size: 98 lines
Goal/features: Shows gang scheduling a simple MPI job on a single machine using the fGangUserAPI.h features.

Fortran Digital MPI (multiple hosts)

Language: Fortran
Sample size: (a) 237 lines, (b) 289 lines
Goal/features: (a) Sample mpichk2.f uses the simple, one-function gs_register() approach, showing the default way to gang schedule a DMPI job.
(b) Sample mpichk.f shows the elaborate (optional) alternative approach in which the entire gang scheduler interface is explicitly added to the code (uses fGangUserAPI.h features and requires SSH support to run).

Fortran90 Threads

Language: Fortran90
Sample size: 26 lines
Goal/features: Shows how to gang schedule a Fortran90 program with four threads (using the fGangUserAPI.h features).

Shared-Memory Parallel Program

Language: C
Sample size: 74 lines
Goal/features: Shows how to gang schedule a shared-memory parallel program written in C by using the GangUserAPI.h features.

Parallel Virtual Machine (multiple hosts)

Language: C
Sample size: 109 lines
Goal/features: Shows how to gang schedule a PVM job across several Digital computers. Both the GangUserAPI.h features (within the code) and SSH support (outside the code) are needed in this case.

<u>Parallel Virtual Machine (multiple hosts)</u>

Language: Fortran
Sample size: 95 lines
Goal/features: Shows how to gang schedule a PVM job across several Digital computers. Both the fGangUserAPI.h features (within the code) and SSH support (outside the code) are needed in this case.

Preemption of Jobs

The Goal.
One way to promote time sharing (running more than one job on a node) is through "concurrent preemption," in which all the tasks of a job are simultaneously suspended (but remain in memory) to make way for another ("expedited") job, then rescheduled as a gang on the same nodes when the other job finishes. Effective preemption requires a way to make and manage preemptable jobs, while still identifying and protecting (a few) specific nonpreemptable jobs. Currently, LC supports job preemption only on its open and secure IBM SP computers (where the gang scheduler is called GangLL).

Preemptable Jobs.
To allow GangLL to preempt a job, that job must be compiled and loaded with thread-safe compilers and libraries. Formerly these had _r in their names (xlf_r, mpxlc_r, libc_r.a), but now such thread-safe software is the default for compiling and loading on LC's IBM SP machines. In fact, whenever preemption is enabled, GangLL will by default treat all jobs in the pbatch class as preemptable. If it encounters a nonpreemptable pbatch job (that has not been protected using the special technique in the next subsection), GangLL will kill the job if it needs to run an expedited job.

When DPCS/LCRM preempts a normal job, then that job:

- halts execution but remains memory resident,
- temporarily releases its nodes for use by the expedited job,
- charges no time (elapsed or CPU) during its preemption pause,
- shows the job status PREEMPTD in PSTAT reports, and
- resumes automatically when the expedited job that borrowed its nodes ends.

Preempting a job alters (prolongs) its apparent run time (or "wall-clock" time). So the current version of DPCS/LCRM incorporates the concept of "interrupted run-time limit," defined as

```
        IRTL = (original run-time limit) + (time spent preempted)
```

to compensate for time spent during preemption. For example, a job with a 2-hour run-time limit that is preempted for 48 minutes will automatically be allowed to stay on the machine for 2 hours and 48 minutes because that longer total time is its "interrupted run-time limit."

Nonpreemptable Jobs.
To protect known nonpreemptable jobs on Blue an extra option (-np) has been added to PSUB. PSUB's -np option works only on SP (IBM) machines, and is ignored for jobs scheduled on any other machines. Invoking -np overrides the job-class (pbatch, pdebug) designation you may have made with PSUB's -c option and places the job in a special "nonstop" class. All nonstop jobs are not time shared, and GangLL does not attempt to run expedited jobs on nodes that are in use by any job in the nonstop class.

To discourage abuse of -np, nonstop nonpreemptable jobs have a maximum time limit of only 2 hours. (Invoking -np has no effect on any other condition that you may have specified with PSUB's -c besides job class.)

There are two special cases of nonpreemptable jobs that do not involve using -np. First, jobs that are already "expedited" are naturally not eligible for preemption. Second, you can make a "job step" of a running job temporarily nonpreemptable (to promote real-time interaction with the job, such as debugging) by using LLEXPRESS (/usr/local/bin/llexpress) on Blue (see its man page for clues).

Preemption Status.
You can determine whether GangLL preemption is enabled or disabled at any time by logging on to Blue and using GREP to check the current value of the EXPEDITE_CLASS variable in the local DPCS configuration file. Here are the steps and their meaning:

```
USER:  grep EXPEDITE /dpcs/adm/pcs.cfg

RTNE: [response]                  [meaning]

      EXPEDITE_CLASS=IGNORE       preemption is
                                  DISABLED
      EXPEDITE_CLASS=EXPEDITE     preemption is
                                  ENABLED
```

To enable such normal-job preemption on an IBM SP (only), the system administrator must (1) set the scheduling mode of LoadLeveler to API, (2) restart the PSPD daemon on the machine's DPCS/LCRM gateway node, and (3) use the LRMMGR utility to specify a suitably large value for "maximum node divergence," the maximum number of nodes that are allowed to go idle as a side effect of scheduling an expedited job over preempted normal jobs. The LRMMGR command to specify maximum node divergence (allowed idle nodes) is

```
    set global maxnodediverge n
```

where *n* is a positive integer.

DPCS Role.
Starting in January, 2001, with version 6.5 of the Distributed Production Control System (DPCS/LCRM), DPCS schedules all IBM SP batch jobs by node pool rather than by job class to better support gang scheduling. Since the IBM SP node pools use names (pbatch, pdebug) formerly associated with job classes on those machines, this technical change should be transparent to users.

How It Works

One gang scheduler daemon excutes on each computer to be scheduled. Programs spanning multiple computers contact the appropriate gang scheduler daemons to be preallocated specific time slices on each computer. An Ousterhout matrix is used to record these preallocated resources as illustrated in this sample table:

	Computer East	Computer East	Computer West	Computer West
	CPU 1	CPU 2	CPU 1	CPU 2
Time 1	Job A	Job A	Job B	Job B
Time 2	Job C	Job C	Job C	Job C
Time 3	Job A	Job A	Job B	Job B
Time 4	Job D	Job D	Job D	Job D

Each processor is represented by one column of the matrix and each row represents one time slice. At prearranged times, the gang scheduler daemons allocate resources as specified in the Ousterhout matrix. The last row in the matrix, time 4, is followed by repeating the cycle from the top, time 1. In this gang scheduler implementation, the Ousterhout matrix describes a one-hour schedule with the first time slice starting on the hour and subsequent time slices at intervals configured when the gang scheduler is built (see below for constraints on the slices).

All computer clocks must be synchronized to within a fraction of a second for concurrent scheduling to occur. LLNL uses a Network Time Protocol (NTP) for clock synchronization, although the Distributed Time Service (DTS) and other systems would be equally satisfactory. The gang scheduler daemon uses an alarm to awake at the appropriate time and runs as user ROOT to avoid being subject to class scheduling constraints.

The gang scheduler is designed to provide each program with access to a similar quantity of processor cycles whether registered for gang scheduling or not. The number of time slices, or entries in the Ousterhout matrix, allocated to a program spanning multiple computers is based upon the load on each computer at program initiation time. The program is allocated a percentage of Ouserhout matrix entries equal to its proportion of threads on the most heavily loaded computer. For example, a program registering with the gang scheduler for four-way parallel on an eight-processor computer with 12 other runnable threads should be allocated 25% of Ousterhout matrix entries on that computer, or four processors every other time slice. A gang scheduler subsystem periodically may increase or decrease the number of time slices preallocated to a program spanning multiple computers as system loads vary.

For programs that run exclusively on one computer, scheduling decisions occur at the beginning of each time slice. These programs lack entries in the Ousterhout matrix, but instead make use of available entries based on current conditions. This permits the gang scheduler to rapidly respond to changed in the workload.

At LLNL, time slices are configured to be relatively long (30 seconds). While such a long time slice reduces program responsiveness, it was required by two factors. First, class scheduler resource allocation targets require on the order of one second propagate to the kernel, resulting in unsatisfactory parallel program overlap for time-slice durations less than about 5 seconds. Second, many programs exceed one gigabyte in size and while context-switching the processor may be performed in milliseconds, the time to refresh the cache may be on the order of hundreds of milliseconds and the time to context-switch memory (paging one program from memory to disk and paging another program in the reverse direction) may be several seconds. To provide faster responsiveness, the execution of a newly initiated program may commence prior to the beginning of a new time slice, if appropriate for the given workload.

Also note that programs not registered for gang scheduling are not subject to these time slices, but are scheduled using normal UNIX scheduling algorithrms and compute resources not allocated to gang-scheduled jobs.

Disclaimer

Keyword Index

To see an alphabetical list of keywords for this document, consult the next section (page 38).

Keyword	Description
entire	This entire document.
title	The name of this document.
scope	Topics covered in this document.
availability	Where these programs run.
who	Who to contact for assistance.
introduction	Gang sched. role, terminology.
usage	When and how to use gang schd.
usage-instructions	How to use gang schd (checklist).
mpi-usage	Using gang schd with MPICH.
dmpi-usage	Using gang schd with DMPI.
pvm-usage	Using gang schd with PVM (full API).
usage-conditions	When to use gang scheduler.
signals	Why gang schd uses signals; disabling.
initialization	(Cray Only) no longer needed.
gangjobinit	(Cray Only) no longer needed.
job-registation	Set resource needs (in C); get job ID.
gangjobregister	Set resource needs (in C); get job ID.
registration-arguments	Job classes for gang schd; other args.
job-classes	Job classes for gang schd; other args.
registration-example	C sample using GangJobRegister.
process-addition	Specify parallel job's processes.
gangprocadd	Specify parallel job's processes.
addition-arguments	How to specify processes.
processes	How to specify processes.
addition-example	C sample using GangProcAdd.
job-monitoring	Tracking parallel jobs with XGANG.
xgang	Tracking parallel jobs with XGANG.
xgang-usage	How to run XGANG monitor.
xgang-machine-details	XGANG display by computer.
xgang-job-details	XGANG display by job.
gang-features	Special gang schd. features.
computer-failure	What happens if a computer fails.
error-codes	Translating gang schd error codes.
gangerrmsg	Translating gang schd error codes.
process-removal	Remove process from a job.
gangprocremove	Remove process from a job.
job-class-change	Changing classes; disabling signals.
signals-disabled	Changing classes; disabling signals.
gangjobclass	Changing classes; disabling signals.
resource-checking	Checking resource availability.
ganghostquery	Checking resource availability.
rpc	Two gang schd remote procedure calls.
event-logging	How to log events (optional).

Alphabetical List of Keywords

Keyword	Description
a	The alphabetical index of keywords.
addition-arguments	How to specify processes.
addition-example	C sample using GangProcAdd.
availability	Where these programs run.
computer-failure	What happens if a computer fails.
date	The latest changes to this document.
dmpi-usage	Using gang schd with DMPI.
entire	This entire document.
error-codes	Translating gang schd error codes.
event-logging	How to log events (optional).
examples	Sample gang scheduler programs.
fortran	Fortran gang-scheduler support.
gang-features	Special gang schd. features.
gangerrmsg	Translating gang schd error codes.
ganggetstats	How to get job-timing info.
ganghostquery	Checking resource availability.
gangjobclass	Changing classes; disabling signals.
gangjobinit	(Cray Only) no longer needed.
gangjobregister	Set resource needs (in C); get job ID.
gangmessage	How to log events (optional).
gangprocadd	Specify parallel job's processes.
gangprocremove	Remove process from a job.
gangresourceregister	Set resource needs in Fortran.
index	The structural index of keywords.
initialization	(Cray Only) no longer needed.
introduction	Gang sched. role, terminology.
job-class-change	Changing classes; disabling signals.
job-classes	Job classes for gang schd; other args.
job-monitoring	Tracking parallel jobs with XGANG.
job-registation	Set resource needs (in C); get job ID.
mpi-usage	Using gang schd with MPICH.
preemption	Job preemption features, options.
process-addition	Specify parallel job's processes.
process-removal	Remove process from a job.
processes	How to specify processes.
pvm-usage	Using gang schd with PVM (full API).
registration-arguments	Job classes for gang schd; other args.
registration-example	C sample using GangJobRegister.
resource-checking	Checking resource availability.
revisions	The complete revision history.
rpc	Two gang schd remote procedure calls.
scope	Topics covered in this document.
signals	Why gang schd uses signals; disabling.
signals-disabled	Changing classes; disabling signals.
threads	Thread use with gang scheduling.
time-slices	How gang schd handles time slices.
timing-information	How to get job-timing info.
title	The name of this document.
usage	When and how to use gang schd.
usage-conditions	When to use gang scheduler.
usage-instructions	How to use gang schd (checklist).

who	Who to contact for assistance.
xgang	Tracking parallel jobs with XGANG.
xgang-job-details	XGANG display by job.
xgang-machine-details	XGANG display by computer.
xgang-usage	How to run XGANG monitor.

Date and Revisions

Revision Date	Keyword Affected	Description of Change
12Nov03	availability	Now on IBM SP machines.
	preemption	Implementation details added.
22Apr02	availability	Gang scheduler on GPS320 only.
	gangjobinit	Call now obsolete.
	entire	CRAY details removed throughout.
10Jan01	preemption	Node pool vs job class scheduling noted.
08Aug00	preemption	Preemption status check explained.
	scope	Print instructions revised.
13Jan00	preemption	New section on job preemption.
15Nov99	threads	Fortran thread-safe library changed.
09Aug99	entire	Expanded and revised throughout.
28Jul99	entire	First edition of Gang Scheduler Guide.

TRG (12Nov03)